Efficient Algorithm for Determining the Optimal Execution Strategy for Path Queries in OODBS

Weimin Chen, Karl Aberer

GMD-IPSI, Dolivostr. 15, 64293 Darmstadt, Germany
E-mail: {chen, aberer}@darmstadt.gmd.de

Abstract. To select an optimal query evaluation strategy for a path query in an object-oriented database system one has to exploit the available index structures on the path. In a database with a large database schema many alternative strategies have to be considered for the evaluation of a path query by choosing from a large set of available indices, which can make the selection of the optimal strategy expensive. We give an algorithm that finds the optimal strategy for evaluating a path query with time complexity independent of the set of indices available in the database. The algorithm considers all possible forward and backward traversal strategies and has time complexity $O(n^2)$ in the path length n. Incorporating this algorithm into the query optimization for object-oriented database management systems can improve the response time of the system, by optimizing an equally important and frequent type of queries with high efficiency.

Key Words. Object-oriented databases, Index selection, Query optimization, Paths

1 Introduction

Object-oriented database systems have a strong relationship to programming languages by means of their data model. Therefore operational aspects play an important role in the access to object-oriented databases. An important kind of operational access is the evaluation of attributes containing object references, i.e. the navigation between objects. The concept of navigation along paths is central for object-oriented database system. With large databases it frequently occurs that large sets of objects are treated in a similar way, in particular the same kind of navigation is performed repeatedly for many objects. When performing such navigations by straightforwardly repeating a single navigation for each object, one restricts the evaluation of such requests to the database basically to a single strategy, for example forward traversal. It was long recognized in relational systems, that processing requests against large sets of objects offers a vast potential for optimizing these requests.

Path queries, i.e. evaluation of paths starting from a set of objects and evaluating a predicate against the value of the path, are an important class of declarative queries. They unify the navigational aspect characteristic for object-oriented database systems and the declarative aspect of uniform access to large object sets. Path queries are one distinctive feature for the usage of object-oriented database systems, that can not be found in relational database systems. Determining the optimal query evaluation strategy for a declarative path query is thus an important problem in the area of object-oriented database systems.

To speed up the evaluation of path queries different indexing mechanisms [4] are proposed in the literature. So an important aspect of the optimization of path queries is the selection of indices on the path. A large database schema may include several hundred classes and attributes, so that the set of index structures defined for the corresponding database can be very large. Different authors [13, 15, 16] discuss how to configure profitable index structures for a given large database schema. With this background, determining the optimal query execution strategy itself can become very expensive for two reasons. Many alternative indices may be available to evaluate the same path and the applicable indices have to be chosen from the whole set of indices, which may be very large. Thus, the *efficient* determination of the optimal strategy to evaluate path queries is another important problem in query optimization. This is important for interpreted query languages and also for query optimization in runtime, as response time is critical. The problem of efficiently selecting a strategy to evaluate paths using indices in an optimal way is still rarely studied.

Bertino *et al*. [14] touches this problem, where several properties to identify a better query execution strategy for paths are discussed. However, an algorithm to generate the optimal execution strategy is still unknown. General purpose approaches, like rule-based query optimization [17], cannot exploit the special structure of the problem, and lead to less efficient search. For example, Kemper and Moerkotte [18] investigate different kinds of rules, that are used to match generalized index structures, so-called access support relations, with path queries. It is not analyzed there, whether the rule system together with their rule interpreter leads to an optimal plan or what is the complexity of the optimization process.

The result of this paper is an efficient algorithm for determining the optimal evaluation strategy for a path query. In particular we exploit the fact that for path query optimization one can optimize each subpath separately, thus avoiding a combinatorial explosion in the search space. We consider forward and backward traversal as well as index evaluation as possible strategies. As the main result we show that we can select the optimal strategy with time complexity independent of the set of available index structures in the database by using a variant of the Aho-Corasick multiple-keyword pattern-matching algorithm [12]. The resulting algorithm has time complexity $O(n^2)$, where n is the length of the query path. For the purpose of comparison we present several alternative algorithms and then compare their time-efficiencies. The results we present can be complemented with other optimization techniques, like those known from relational query optimization, or techniques, that consider other important aspects of the object-oriented data model, like the semantic optimization of methods in queries [10].

The remainder of this paper is organized as follows: Sections 9 reviews several basic concepts with respect to paths and query processing. Section 10 summarizes the main principles of the indexing techniques considered in this paper. Sections 11, 12, and 13 discuss several algorithms for determining the optimal query execution strategy. Finally, we conclude in Section 14.

2 Basic Concepts

A *query path* is specified by $P = C_1.A_1.A_2. \ldots .A_n$ where A_i are attributes of a class C_i, and the value of A_i is an object of class C_{i+1} ($1 \leq i < n$). We assume that A_i is globally identified.[1] With this assumption, a query path P can be simply represented by $A_1.A_2. \ldots .A_n$. For a globally identified attribute A, let *class*(A) denote the class in which A is defined. Thus, in path P the type of A_i is *class*(A_{i+1}), for $i < n$. In the following, a subpath $A_i.A_{i+1}. \ldots .A_j$ ($1 \leq i \leq j \leq n$) of P is denoted by $P(i, j)$.

In the following let a path $P = A_1.A_2. \ldots .A_n$. A *complete path instantiation* of P is a sequence $o_1.o_2. \ldots .o_{n+1}$ such that o_1 is an instance of *class*(A_1) and o_{i+1} the value (contained in the value) of attribute A_i of object o_i, $1 \leq i \leq n$. A *partial instantiation* of P is a complete path instantiation of a subpath $P(i, n)$, $1 < i \leq n$. Furthermore, a partial instantiation $o_i.o_{i+1}. \ldots .o_{n+1}$ of P is called *non-redundant* if there does not exist an object o such that $o.o_i.o_{i+1}. \ldots .o_{n+1}$ forms another partial instantiation of P.

The *value of path P for object o* is the value o_{n+1} of a complete path instantiation $o.o_2. \ldots .o_{n+1}$ of P. A path *query* (P, V) is specified by a path P and a set of values V. Sometimes the set V is specified by a predicate. The *result of a query* (P, V) is the set of all objects of *class*(A_1) for which the value of P is contained in V.

A query (P, V) can be split into the following subqueries processed in the right-to-left order:

$$(P(i_0, i_1 - 1), V_1),\ (P(i_1, i_2 - 1), V_2),\ \ldots,\ (P(i_j, i_{j+1} - 1), V_{j+1}),\ \ldots,\ (P(i_k, i_{k+1} - 1), V_{k+1}) \quad (1)$$

where $1 = i_0 < i_1 < \ldots < i_{k+1} = n + 1$, $V_{k+1} = V$, and V_j ($1 \leq j \leq k$) is the set of objects of *class*(A_{i_j}) which is the result of the subquery ($P(i_j, i_{j+1} - 1)$, V_{j+1}). Clearly, V_j is also the result of the query ($P(i_j, n)$, V). For each above subquery ($P(i_j, i_{j+1} - 1)$, V_{j+1}), $0 \leq j \leq k$, three kinds of query execution strategies are considered in this paper:

- *Forward traversal*: Evaluate for all objects of *class*(A_{i_j}) the value of the path $P(i_j, i_{j+1} - 1)$ and check whether it is contained in V_{j+1} (note, that V_{j+1} is already known, as subqueries (5) are processed in right-to-left order). The cost of evaluating the path $P(i_j, i_{j+1} - 1)$, by forward traversal can be computed in constant time. The corresponding cost function depends on some parameters, e.g. the cardinality of class *class*(A_{i_j}) associated with attribute A_i and the value $i_{j+1} - i_j$.

- *Backward traversal:* Some OODBSs can support inverse references; with these it is possible to traverse the path from right to left starting from all objects in V_{j+1} for the query ($P(i_j, i_{j+1} - 1)$, V_{j+1}). The cost of backward traversal for a path $P(i, j)$, $1 \leq i \leq j \leq n$, depends on a number $k_i \times k_{i+1} \times \ldots \times k_j$, where k_i is the average number of instances of *class*(A_i) assuming the same values for attribute A_i. In the algorithms presented in this paper, the cost of backward traversal for path $P(i, j)$ is always calculated after the cost for path $P(i + 1, j)$ is evaluated. Thus, by means of an incremental algorithm, the cost of backward

2. This can be easily achieved by re-identifying the attribute A as $C::A$ where A is the attribute of class C.

traversal for path $P(i, j)$ can be calculated in constant time, based on the already calculated cost for path $P(i + 1, j)$.

- *Indexed access*: For each value in V_{j+1}, an index (if any exists) matching the path $P(i_j, i_{j+1} - 1)$ is evaluated, by which the result is returned. Once a particular index is identified to match a path $P(i, j)$ the index access cost can be evaluated in constant time.

The cost of evaluating a subquery $(P(i_j, i_{j+1} - 1), V_{j+1})$ relies on what query strategy is chosen as well as on the cardinalities of the sets V_j and V_{j+1}, but it is independent of the strategies chosen for the previous subqueries split from $(P(i_{j+1}, n), V)$. This is the so-called *separability* property [13]. It is important because the optimal query execution strategy can be chosen for each subpath independently from the query execution strategies chosen for the other subpaths. Without that, the optimal query strategy would have to be determined in exponential time in general.

3 Index Organizations

In this paper, two kinds of indices, *path index* and *nested index* [14], as well as any collection of path indexes and nested indexes for subpaths of a path, a so-called *multi-index*, is considered. In the following we use the definitions from [13].

Given a path $P = A_1.A_2. \ldots .A_n$, a path index on P is a set of pairs $(o_n, o_j.o_{j+1}. \ldots .o_n)$, $j \geq 1$, where $o_j.o_{j+1}. \ldots .o_n$ is either a complete instantiation or a non-redundant partial instantiation of P. A nested index on P is a set of pairs (o_n, o_1) where there exists a complete instantiation of P equal to $o_1.o_2. \ldots .o_n$. For both kinds of indices, the first elements of the above pairs are the index keys. Thus, given a path P a nested index defined on P supports efficient execution of the query on P, while a path index defined on P supports efficient execution of each query $(P(i, n), V)$, $1 \leq i \leq n$. That is, a single path index can support queries on n different paths.

Logically, we treat a path index defined on $P = A_1.A_2. \ldots .A_n$ as n different indices each of which directly supports a query on the corresponding subpath $P(i, n)$, $1 \leq i \leq n$. Thus, we can make a correspondence between a logical index (i.e. a nested index or a logical path index) and a path on which the index can directly support the execution of queries. In the rest of this paper, we always assume that the indices under discussion are logical.

By this correspondence, the *length* of a logical index I, denoted by $len(I)$, is defined as the length of the path corresponding to the index. For convenience, an index I corresponding to path $A_1.A_2. \ldots .A_n$ is sometimes denoted by $I(A_1A_2 \ldots A_n)$.

In this paper, let $\mathcal{I}$ denote the set of all available logical indices. We will always assume that *all indices in $\mathcal{I}$ correspond to different paths* for the following reason: obviously, it makes no sense to maintain two path indices or two nested indices for the same path. It might in exceptional cases be reasonable to maintain a nested index for a path that is also covered by a (logical) path index. In this case, however, we assume that execution of the nested index is always cheaper than execution of the logical path index on the same path, thus we eliminate the logical path index from $\mathcal{I}$. For example, we define

$$\mathcal{I} = \{I_1(BCDE), I_2(ABC), I_3(BC), I_4(C)\},$$

where I_1 is a nested index, I_2 and I_4 are two logical path indices derived from a physical path index for path *ABC*, and I_3 is a nested index. Note that we did not include the logical path index for subpath *BC*, because it is already covered by the nested index. We have $len(I_1) = 4$, $len(I_2) = 3$, $len(I_3) = 2$, and $len(I_4) = 1$.

Given a query path P and an index $I(A_1A_2...A_l)$, if $P(j - l + 1, j) = A_1.A_2. ... .A_l$ we say *I matches on P at position j*. If $P = A_1.A_2. ... .A_l$ we say *I matches on P.*

4 Splitting Algorithm

For the purpose of comparison, this section presents a preliminary algorithm for the selection of a query execution strategy. The algorithm is derived from [13]. Originally the algorithm is applied to the index configuration problem, but it can be directly used for the query execution strategy selection problem with a slight modification.

The algorithm is organized in n steps: in the l^{th} step all subpaths of length l are considered. For each subpath, $P(i, i + l - 1)$ say, the algorithm considers the following $l + 2$ strategies:

- The first two strategies are forward and backward traversal.
- The third strategy is an indexed access if an index in $\mathcal{I}$ matches on $P(i, i + l - 1)$.
- The remaining $l - 1$ strategies are obtained by splitting the subpath $P(i, i + l - 1)$ into two subpaths $P(i, i + k)$ and $P(i + k + 1, i + l - 1)$, where $0 \le k < l - 1$. For each split, the cost of the path $P(i, i + l - 1)$ is given as the sum of the costs of the two subpaths. Note that the cost of the two subpaths has already been evaluated at a previous step since their lengths are strictly less than l.

In each step the most cost efficient of the $l + 2$ strategies is selected. Thus, in the n^{th} step the most cost efficient strategy for the whole path P is determined.

In the following, let δ_i^l denote the cost of the most efficient query execution strategy on subpath $P(i, i + l - 1)$ with length l. Thus, δ_1^n is the cost of the most efficient query execution strategy for the path $P = P(1, n)$.

```
routine splitting
1  {   for l = 1 to n do
2          for i = 1 to n – l + 1 do
3          {   find index I_q ∈ ℐ matching on P(i, i + l – 1)
4              if found then cost_index ← cost_idx(I_q, i + l –1)
5                       else cost_index ← ∞
6              δ_i^l ← min ({cost_fwd(i, i + l –1), cost_bwd(i, i + l –1), cost_index} ∪
                           {δ_i^k + δ_{i+k}^{l-k} | 1 ≤ k < l})
7          }
8  }
```

In the above algorithm, $cost_{fwd}(i, j)$ and $cost_{bwd}(i, j)$ represent the costs for evaluating the query using forward and backward traversal on subpath $P(i, j)$, and $cost_{idx}(I_q, j)$ represents the cost of the indexed access where I_q matches on path P at position j.

When no backward traversal can be supported on subpath $P(i, j)$, let $cost_{bwd}(i, j) = \infty$. Note that the assumption made in Section 9 on the computation of the cost for backward traversal is satisfied in this algorithm since the costs for query execution strategies for the subpaths of a path are always calculated in advance.

Determining which index can match on $P(i, j)$ will take average $O(j - i)$ time for symbol comparisons.

On the other hand, calculating the minimum for δ_i^l (line 13) needs $l + 2$ steps for comparisons. Thus, the total average time cost of the above algorithm is

$$O\left(\sum_{l=1}^{n} l(n - l)\right) + O\left(\sum_{l=1}^{n}(n - l + 1)\right) + O\left(\sum_{l=1}^{n}(l + 2)(n - l + 1)\right) \quad (2)$$

where the first term is the cost for index matching done in line 11, the second the cost for calculating $cost_{fwd}(i, j)$ and $cost_{bwd}(i, j)$, and the third the cost for calculating the minimum done in line 13. Clearly, expression (6) is equal to $O(n^3)$.

5 Right-to-Left Matching Algorithm

In this section we present an algorithm by right-to-left matching, that is similar to the dynamic programming strategy used for example by the rule-based *Volcano* query optimizer generator [17] for obtaining optimal plans. The idea of the algorithm is to recursively determine the query execution strategies with the minimal cost on the subpaths $P(i, n)$, $1 \leq i \leq n$. Finally, the optimal query execution strategy can be generated on the path $P = P(1, n)$.

For a given query path $P = A_1.A_2. \ldots .A_n$, we create an array, $a[1 .. n]$, to perform index matching, where each entry $a[i]$ $(1 \leq i \leq n)$ is a record consisting of the following items:

- $a[i].cost$: The cost of the optimal query strategy on path $P(i, n)$;
- $a[i].strategy \in \{indexed\_access, forward\_traversal, backward\_traversal\}$: The query strategy chosen for the left-most subquery of the optimal query strategy for $P(i, n)$;
- $a[i].subscript$: The subscript of the index I_q which is chosen for the left-most subquery of the optimal query strategy for $P(i, n)$, when $a[i].strategy = indexed\_access$;
- $a[i].position$: The position j of the subpath $P(i, j)$ which corresponds to the left-most subquery (forward or backward traversal) of the optimal query strategy on path $P(i, n)$, when $a[i].strategy \in \{forward\_traversal, backward\_traversal\}$.

The following routine $match_1$ traverses P from right to left in order to perform the minimal cost matching. Initially, before the call of $match_1$, let $a[i].cost = \infty$ for $1 \leq i \leq n$.

```
routine match1()
1   {   for i = n downto 1 do
2       {   find all possible indices I_q1, I_q2..., I_qk matching on P at position i
3           foreach above matched index I_q do idx_match(i, q)
4           for k = i downto 1 do nonidx_match(k, i)
5       }
6   }
```

In the above routine, two subroutines are invoked: subroutine *idx_match*(i, q) determines the cost of a subquery when an indexed access by I_q matching on P at position i is considered; subroutine *nonidx_match*(k, i) determines the cost of the subquery when the forward or backward traversal on subpath $P(k, i)$ is considered.

```
subroutine idx_match(i, q)
1   {   new_cost ← a[i + 1].cost + cost_idx(I_q, i)      /* assume here a[n + 1].cost = 0 */
2       k ← i – len(I_q) + 1
3       if a[k].cost > new_cost then
4       {   a[k].cost ← new_cost
5           a[k].strategy ← indexed_access
6           a[k].subscript ← q
7       }
8   }
```

In the above algorithm, function $cost_{idx}(I_q, i)$ returns the retrieval cost of indexed access when index I_q matches on P at position i.

```
subroutine  nonidx_match(k, i)
1   {   cost_fwd ← cost_fwd(k, i)
2       cost_bwd ← cost_bwd(k, i)
3       new_cost ← min{cost_fwd, cost_bwd} + a[i + 1].cost
            /* assume here a[n + 1].cost = 0 */
4       if a[k].cost > new_cost then
5       {   a[k].cost ← new_cost
6           a[k].strategy ← if cost_fwd < cost_bwd then forward_traversal
                            else backward_traversal
7           a[k].position ← i
8       }
9   }
```

In routine $match_1$, we have to determine all possible indices matching on path P at position i (line 2). Thus we have to check all indices of length less than i. This needs average $|\mathcal{I}_i|/2$ times for symbol comparisons, where $\mathcal{I}_i$ is the set of indices I in $\mathcal{I}$ such that $len(I) \leq i$.

Let N_{IDX} denote the total number of invocations of subroutine *idx_match*. It is easy to check that $N_{IDX} \leq n(n + 1)/2$, where $n(n + 1)/2$ is the number of subpaths of

P. In general $N_{IDX} << n(n+1)/2$, as not for every subpath an index exists. On the other hand, the number of invocations of subroutine *nonidx_match* is $n(n+1)/2$.

As both subroutines *idx_match* and *nonidx_match* can be executed in constant time, let $C_{idxmatch}$ and $C_{nonidxmatch}$ denote the corresponding time costs. The average time cost of $match_1$ is

$$C_{sym\_cmp} \cdot \left(\sum_{i=1}^{n} |\mathcal{I}_i|/2 \right) + C_{idxmatch} \cdot N_{IDX} + C_{nonidxmatch} \cdot n(n+1)/2 \qquad (3)$$

where C_{sym_cmp} is the time cost for a symbol comparison.

In a large OODBS, the set of indices $\mathcal{I}$ can be very large, so that the value $\sum_{i=1}^{n} |\mathcal{I}_i|/2$ can be very large too. Thus, in realistic applications, where n is between 1 and 10, the value of expression (7) would be mainly determined by the first term for a large set $\mathcal{I}$. In the next section, we will present an improvement such that the time cost is fully independent of $\mathcal{I}$.

Once the above minimal cost matching is performed, the optimal execution strategy can be obtained by a new routine *output_query_strategy* which traverses the array *a* in the left-to-right order.

```
routine output_query_strategy()
{   i ← 1
    while i < n do
        if a[i].strategy = indexed_access then
        {   output the index I_{a[i].subscript}
            i ← i + len(I_{a[i].subscript})
        }
        else /*a[i].strategy = forward_traversal or backward_traversal */
        {   output  forward (or backward) traversal for path P(i, a[i].position)
            i ← a[i].position + 1
        }
    }
}
```

Clearly, the time cost of the above routine is $O(n)$.

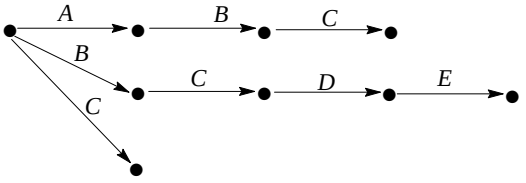
6 Efficient Index Matching Algorithm

In this section we present an improvement to further reduce the time cost of routine $match_1$. More concretely, we want to reduce the time cost of finding all indices matching on path *P* at some positions. As indicated in (7), this cost is $C_{sym\_cmp} \cdot (\sum_{i=1}^{n} |\mathcal{I}_i|/2)$.

The algorithm we suggest is based on the Aho-Corasick multiple-keyword pattern-matching algorithm [12]. First consider the problem of finding all substrings of an input string that are contained in a given set of keywords. The essence of the Aho-Corasick algorithm is to construct a trie [11] from the set of keywords, convert the trie

into a pattern-matching automaton, and then use the pattern-matching automaton to perform a parallel search for the keywords in the input string.

Let *K* be the set of keywords. The trie is built by first making a root node and then, for each keyword in *K*, creating a path from the root to a node whose branch labels spell out the keyword. Each node of the trie is thus uniquely characterized by the sequence of symbols on the branch labels of the path from the root to that node. An example of a trie is shown as follows for the set $K = \{BCDE, ABC, BC, C\}$:



The pattern-matching automaton is constructed from the trie. The states of the automaton are the nodes of the trie; the start state is the root and the accepting states are those corresponding to complete keywords. There is a transition from state σ_1 to σ_2 on input character *c* if there is a branch in the trie labeled *c* from node σ_1 to node σ_2. Furthermore, we add a transition from the start state to itself on every input character that is not the first character of a keyword.

The pattern-matching automaton has a failure function for every state other than the start state. The failure function for a state characterized by a string *u* is a pointer to the state characterized by the longest prefix of some keyword in *K* that is also a proper suffix of *u*.

Both the trie and the pattern-matching automaton can be constructed in time linearly proportional to the sum of the lengths of the keywords in *K*. The resulting pattern-matching automaton can be run on an input string *x* in time linearly proportional to the length of *x*, independent of the size of *K*.

This algorithm can be directly applied for the query strategy decision algorithm by noticing that each index can be characterized by a path-string (each symbol in the string stands for a globally identified attribute). Using the Aho-Corasick algorithm, we can construct a pattern-matching automaton to match the path-strings in parallel.

Fig. 3 shows the pattern-matching automaton corresponding to the set of path-strings and their corresponding indices from our example given in Section 10. State 0 is the start state and the double-circled states are accepting. In this automaton the failure function for state 2 points to state 4, for state 3 to state 5, and for all other states to state 0; the failure functions that do not point to state 0 are shown as dashed lines. At each accepting state, we also know which keywords and which indices have been recognized. For example, at state 3 the recognized indices are I_2, I_3, and I_4; at state 7 the recognized index is I_1.

Let $succ(\sigma, \alpha)$ denote the state reached from the state σ on input symbol α by the automaton. For example, in the automaton shown in Fig. 3, $succ(4, C) = 5$, $succ(4, D) = 0$, and $succ(3, D) = 6$.

By means of the pattern-matching automaton, the query strategy decision algorithm can be processed more efficiently. Recall that, in the algorithm $match_1$, the array $a[1 .. n]$ corresponding to a query path $P = A_1.A_2. ... .A_n$ is created where each

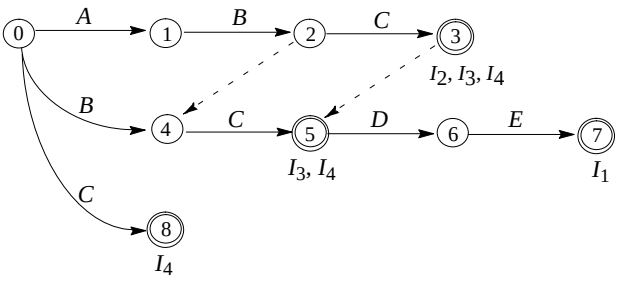


Fig. 1 The pattern-matching automaton for the set of indices $\mathcal{I} = \{I_1(BCDE), I_2(ABC), I_3(BC), I_4(C)\}$

entry $a[i]$, $1 \le i \le n$, is a record. The following routine $match_2$ performs an efficient index matching, based on the same array a but the entry $a[i]$ has an additional item:

- $a[i].state$ to indicate the state of the automaton after it visits $a[i]$.

The following routine $match_2$ will first traverse query path P to assign a state to each entry of a in the left-to-right order, and then perform the minimal cost matching in the right-to-left order.

Initially, let $a[i].state = 0$ and $a[i].cost = \infty$ for $1 \le i \le n$ before $match_2$ is called.

```
routine match2()
1  {   a[1].state ← succ(0, A1)
2      for i = 2 to n do a[i].state ← succ(a[i–1].state, Ai)
3      for i = n downto 1 do
4      {   if a[i].state is accepting then
5              foreach index Iq recognized at a[i].state do idx_match(i, q)
6          for k = i downto 1 do nonidx_match(k, i)
7      }
8  }
```

According to [12], the number of state transitions (including the failure state transitions) is not greater than $2n$. Thus, the time cost is $match_2$ is

$$C_{transition} \cdot 2n + C_{idxmatch} \cdot N_{IDX} + C_{nonidxmatch} \cdot n(n+1) \,/\, 2 \qquad (4)$$

where $C_{transition}$ is the cost of state transition, $C_{idxmatch}$ and $C_{nonidxmatch}$ are the costs of executing functions *idx_match* and *nonidx_match*.

Comparing expression (8) to (7), the time cost of $match_2$ is in general reduced essentially in (8), since the term $C_{sym\_cmp} \cdot (\sum_{i=1}^{n} |\mathcal{I}_i| / 2)$ in (7) has been replaced by term $C_{transition} \cdot 2n = O(n)$ in (8). Thus, the time cost of routine $match_2$ is $O(n^2)$ and is completely independent of the structure of $\mathcal{I}$.

For example, consider the query path $P = A.B.C.D.E.F$. Based on the set $\mathcal{I}$ and the corresponding automaton as given in Fig. 3, applying $match_2()$ yields the values at each entry shown in Fig. 4.

	1	2	3	4	5	6
1	1	2	3	4	5	6
2		1	2	3	4	5
3			1	2	3	4
4				1	2	3
5					1	2
6						1

(a) Table of $cost_{fwd}$

	1	2	3	4	5	6
1	1	∞	∞	∞	∞	∞
2		∞	∞	∞	∞	∞
3			∞	∞	∞	∞
4				∞	∞	∞
5					∞	∞
6						∞

(b) Table of $cost_{bwd}$

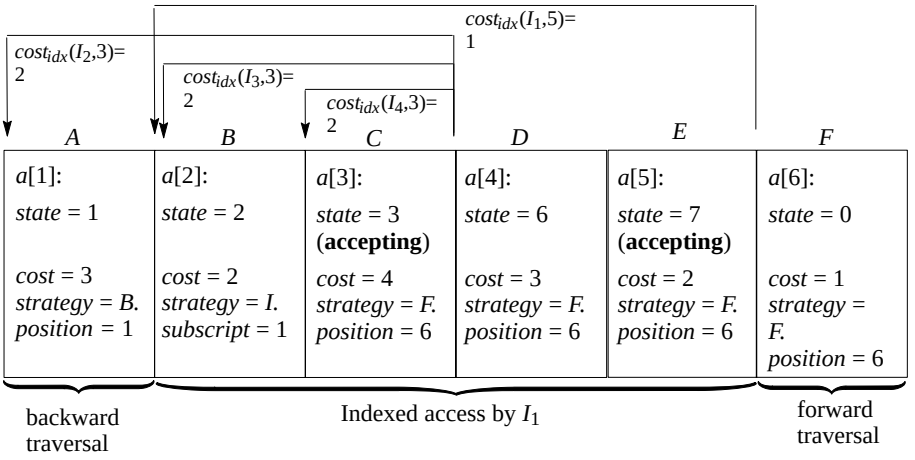


Fig. 2 Pattern-matching found by routine $match_2$

In Fig. 4, we see that the cheapest cost at entry $a[1]$ is 3; by the routine *output_query_strategy* described in Section 12 the optimal query strategy in the right-to-left order is

(a) forward traversal on subpath $P(6, 6) = F$,
(b) indexed access on subpath $P(2, 5) = B.C.D.E$, and
(c) backward traversal on subpath $P(1, 1) = A$.

Building the pattern-matching automaton leads to an additional overhead. However, the pattern-matching automaton must be rebuilt only when index structures are changed, i.e. indices are added to or deleted from $\mathcal{I}$. In general the frequency of updating the index structures is very low in comparison to the frequency of queries. Thus, the benefits of introducing the pattern-matching automaton are greater than the cost of the maintenance of the pattern-matching automaton.

7 Conclusions

We have presented an efficient algorithm for determining the optimal execution strategy for path queries in OODBSs. We adopted the Aho-Corasick multiple-keyword

pattern-matching algorithm and combined it with a right-to-left matching strategy for obtaining the optimal plan. The time-efficiency of our algorithm is independent of the set of indices and achieves time complexity $O(n^2)$ in the length n of the path query. The paper provides a careful analysis of the complexity of the problem. This analysis shows that for moderate path lengths the crucial factor for time complexity of our problem is in large databases the cardinality of the set of available indices. By eliminating the dependency on this we can avoid the crucial bottleneck.

This technique covers an important aspect of declarative access to object-oriented databases and thus complements other query optimization techniques for declarative object-oriented query languages. An open issue is to extend this technique to query trees, i.e. queries that contain multiple predicates on paths. We consider this to be an extremely hard problem, because in this case the number of alternative query strategies is combinatorially exploding.

References

1. ABERER, K., AND FISCHER, G. Semantic Query Optimization for Methods in Object-Oriented Database Systems. *Proc. of the 11th ICDE*, 1995.
2. AHO, A. V., HOPCROFT, J., AND ULLMAN, J. D. Data Structures and Algorithms, *Addison-Wesley Publishing Company.*
3. AHO, A. V. AND CORASICK, M. J. Efficient string matching: An aid to bibliographic search. *Comm. of ACM* 18,6, June 1975.
4. BERTINO, E. Index Configuration in Object-Oriented Databases. *VLDB Journal* 3,3 1994.
5. BERTINO, E. AND GUGLIELMINA. Path-Index: An Approach to the Efficient Execution of Object-Oriented Queries. *Data & Knowledge Enginerring 10, North-Holland, 1993.*
6. CHAWATHE, S. S., CHEN M.-S., AND YU, P. S. On Index Selection Schemes for Nested Object Hierarchies. *Proc. of the 20th VLDB*, 1994.
7. CHOENNI, S., BERTINO, E., BLANKEN, H. M., AND CHANG, T. On the Selection of Optimal Index Configuration in OO Databases. *Proc. of the 10th ICDE*, 1994.
8. GRAEFE, G. AND MCKENNA W. J. Extensibility and Search Efficiency in the Volcano Optimizer Generator, *Proc of the 9th ICDE*, 1993.
9. KEMPER, A., AND MOERKOTTE, G. Advanced Query Processing in Object Bases Using Access Support Relations, *Proc. of the 16th VLDB*, 1990.